SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Alen Huskanović

METODE TESTIRANJA IZVORNOG KODA MOBILNIH APLIKACIJA ZA OPERACIJSKI SUSTAV ANDROID

DIPLOMSKI RAD

Varaždin, 2018.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Alen Huskanović

Matični broj: 43510/14-IZV

Studij: Informacijsko i programsko inženjerstvo

METODE TESTIRANJA IZVORNOG KODA MOBILNIH APLIKACIJA ZA OPERACIJSKI SUSTAV ANDROID

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Neven Vrček

Varaždin, srpanj 2018.

Sadržaj

1.Uvod

S povećanjem globalizacije, razvojem interneta te napretkom razvoja tehnologije kao što su računala i mobilni uređaji industrija razvoja programskih rješenja je eksponencijalno narasla. To su iskoristile mnoge organizacije pa je nastalo cijelo tržište kompanija koje po narudžbi razvijaju programska rješenja za firme od pekarna pa sve do velikih konzultantskih korporacija. Prisjetimo li se samo devedesetih godina kada su tehnološke kompanije, koje su se uglavnom bazirale na razvoju aplikacija na internetu, toliko rasle da im kapitalizacija jednostavno nije imala smisla pa se dogodio dot.com crash tržišta. Mnogi su tada govorili da je ta industrija gotova, no danas je nemoguće zamisliti business koji funkcionira bez da ima digitalnu prisutnost na internetu.

U to vrijeme su uglavnom kompanije imale fokus na programskim rješenjima koja su se nalazila na internetu te im se pristupalo preko računala, no s razvojem pametnih telefona je sve više kompanija krenulo u razvoj mobilnih programskih rješenja koja su fleksibilnija i dostupna na svakom pametnom telefonu. Tržište digitalnih kompanija koje se u procesu razvilo je postalo toliko zasićeno da se na dnevnoj bazi osnivaju nove kompanije koje razvijaju programska rješenja ali i dalje je potražnja veća od ponude. Iako je takvo tržište zdravo, dovelo je do problema da kompanije uzimaju razne projekte iako nemaju dovoljno ljudstva da ih kvalitetno odrade pa sada na tržištu postoji veliki broj aplikacija koju nisu stabilne i imaju mnogobrojne greške u radu.

Da bi se stalo na kraj nekvalitetnim rješenjima i konstantnim pogreškama u radu raznih aplikacija, klijenti su počeli tražiti nove pristupe s kojima će im izvršitelj moći garantirati da njihovo programsko rješenje radi točno na način kako je to zamišljeno. Kao odgovor tome (ali i mnogim drugim problemima) se uvodi pojam testiranja programskih rješenja, a u zadnjih nekoliko godina taj novi način preuzima ulogu te postaje standard ne samo u digitalnim agencijama, već u svim kompanijama koje razvijaju nekakvo programsko rješenje.

U ovom radu će se obraditi principi i načini testiranja programskih rješenja za operacijski sustav Android. Osim teoretskog dijela, u radu će biti opisan i proces uvođenja procesa testiranja u postojeću aplikaciju za Android operacijski sustav.

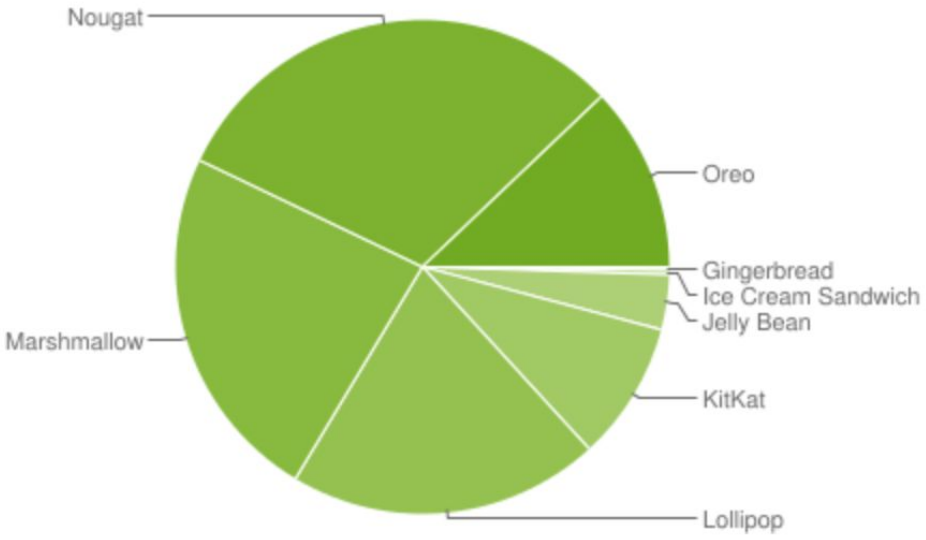
2. Operacijski sustav Android

Trenutno na tržištu postoji nekoliko operacijskih sustava koji se koriste na pametnim telefonima, ali zbog premalog obujma na tržištu mnogi od njih nisu vrijedni spomena, pa navodimo samo one koji imaju reprezentativan udio tržišta.

Prema Statecounter statistici, trenutno Android drži udio od 77.32% tržišta, dok je odmah iza njih s velikom razlikom iOS koji ima tek 19.4% tržišta, nakon njega imamo Windows koji zauzima tek mali postotak od 0.43%. Svi ostali operacijski sustavi imaju premali udio tržišta pa neće biti spomenuti u ovome radu. [3]

Kao što se da primijetiti na statistici, Android drži gotovo 80% tržišta, te je samim time najpopularniji operacijski sustav korišten za pametne telefone, tablete i slične uređaje. Osim na pametnim telefonima, Android možemo pronaći u raznim kućanskim aparatima kao što su mikrovalne pećnice, klima uređaji, hladnjaci itd. Android je u početcima razvijan od firme Android Inc, koju je Google kupio 2005, te je razvoj nastavljen i u konačnici je objavljen od strane Open Handset Alliance (OHA) što je udruženja 84 firme koje su se borile za razvoj operacijskog sustava za mobilne platforme pod standardima otvorenog kôda. Android je objavljen odmah po uspostavljanju OHA 5. 11. 2007 godine.

Prema zadnjim podacima od Googlea, najpopularnija inačica Androida je Marshmallow s impresivnih 23.5%, no bitno je napomenuti da je to Android verzija 6.0. dok je trenutno zadnja objavljena verzija Oreo, tj. 8.0. [4]



Slika 1. Grafički prikaz popularnosti Android verzija

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.2%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.3%
4.1.x	Jelly Bean	16	1.2%
4.2.x		17	1.9%
4.3		18	0.5%
4.4	KitKat	19	9.1%
5.0	Lollipop	21	4.2%
5.1		22	16.2%
6.0	Marshmallow	23	23.5%
7.0	Nougat	24	21.2%
7.1		25	9.6%
8.0	Oreo	26	10.1%
8.1		27	2.0%

Slika 2. Tablični prikaz popularnosti Android verzija

Android operacijski sustav je softver koji je u potpunosti otvorenog kôda, a baziran je na Linux kernelu te je pušten pod Apache licencom.

Google Play Store je naziv trgovine za prodaju i distribuciju mobilnih aplikacija napravljenih za Android operacijski sustav, a omogućuje vrlo jednostavan, jeftin i brz proces objave aplikacije. Za razliku od Applea storea, Google Play Store je daleko jednostavniji i jeftini za objavu aplikacije pa to privlači veliki broj programera koji zatim razvijaju aplikacije za ovu platformu. Ovo je s jedne strane dobra stvar, no s druge strane ja zapravo loše jer je Google Play Store preplavljen lošim aplikacijama.

4. Test Driven development

Test driven development (TDD) se odnosi na pristup testiranju u kojem se testovi pišu prije konkretnog koda. Točnija definicija bi bila da je test driven development praksa u kojoj programeri pišu za produkciju tek nakon što su napisali automatizirane testove. [1]

Ovakva praksa kod razvoja softvera se popuno razlikuje od standardne prakse gdje se testovi pišu nakon što je napisati izvorni kod kako bi se testiralo radi li baš ono za što je namijenjen. U TDD praksi je vrijeme pisanja testova na popuno drugom mjestu. Ovakav pristup testiranju se počinje primjenjivati u devedesetim godinama prošlog stoljeća kada je Kent Beck, autor knjige Extreme Programming Explained: Embrace Change spominjao takav pristup dok je razvijao aplikacije u Smalltalk kompaniji, iako je i sam Kent u knjizi napomenuo da ideja za pisanjem testova prije izvornog koda nije potekla od njega, te se nadovezao na NASU, koja je koristila inicijalnu verziju TDD pristupa 1960-te godine pri razvoju softvera za rakete.

TDD u to vrijeme nije bio detaljno specificiran. te su se programeri vodili samo s jednim pravilom, a to je da nikada ne napišu liniju izvornog koda prije nego su napisali testove.

Isti autor, Beck je nekoliko godina kasnije izdao još jednu knjigu u kojoj se doticao TDD pristupa, a u njoj je cijelu praksu sveo u nekoliko koraka:

1. Napisati novi testni use case
2. Izvršiti sve testove i vidjeti hoće li novi test biti jedini koji neće proći
3. Napisati izvorni kod dovoljno da test prođe
4. Ponovno izvršiti sve testove i vidjeti hoće li sada svi testovi proći
5. Refakturirati izvorni kod tako da se maknu sve nepotrebne linije

Problem kod ovoga pristupa je da nije u potpunosti prihvaćen od strane programera, jer većina njih ne vidi korist u tome pristupu i smatraju vrijeme utrošeno u pisanje testova

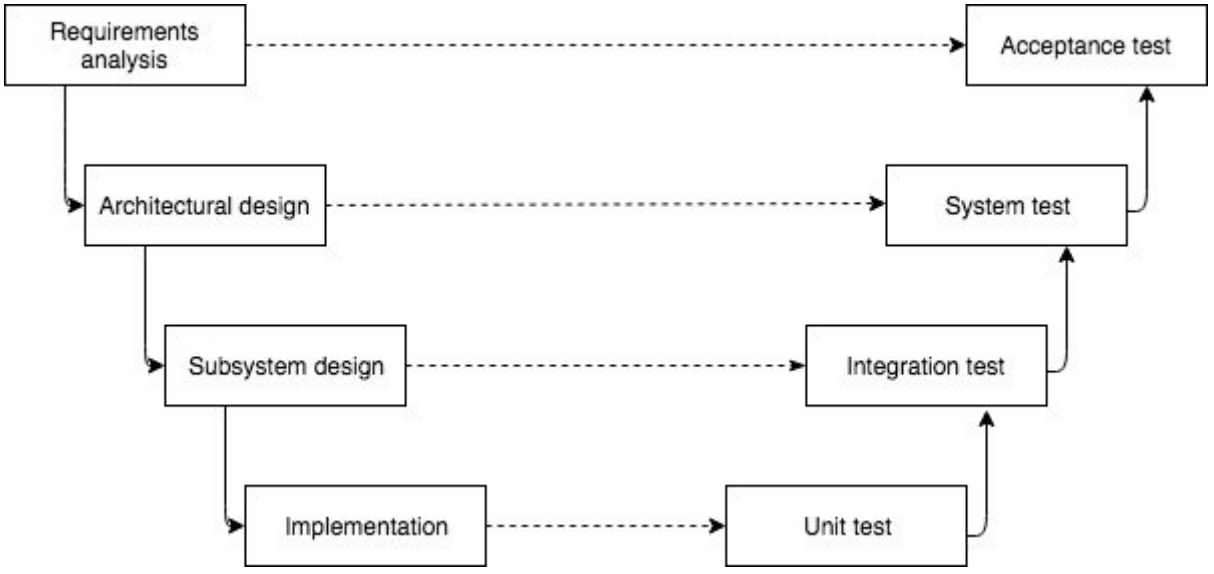
nepotrebno. Iz tog razloga se u ovome radu nećemo držati ovakvoga pristupa već ćemo se fokusirati na uvođenje testova nakon što je već napisan izvorni kod.

5. Razine testiranja softvera

Testiranje softvera je široki pojam, te postoje razne područja koja se mogu testirati. Neke od mogućnosti su testiranje zahtjeva i specifikacije softvera, testiranje dizajna i sučelja, testiranje ponašanja izvornog koda itd… U ovome radu ćemo obraditi dvije različite podjele testiranja: testiranje bazirano na aktivnostima softvera, te testiranje bazirano na zrelosti procesa testiranja.

5.1 Podjela testova baziranih na programskim aktivnostima

Kod uvođenja procesa testiranja u proces razvoja softvera testovi se grupiraju u određene levele prema njihovoj svrsi i konkretnom cilju testiranja. Najčešća podjela testova se odvija u tri levela, a to su unit testovi, integracijski testovi i sistemski testovi. Postoji još razina testova prihvatljivosti (engl. acceptance testing), no ono je odvojeno od glavne tri razine jer često nisu programeri odgovorni za provođenje tih testova.



Slika 3. Razine testiranja softvera

Unit testing su testovi koji služe za testiranje konkretne implementacije, odnosno testovi koji testiraju izvorni kod. Ova razina testiranja je najniža razina testova, te je najbitnija ako želimo testirati linije koda, te metode u određenim klasama. Ovakvi testovi se u pravilo pišu na razini specifikacije gdje svaki mali zadatak tijekom implementacije treba imati svoj unit test. Ovi testovi su pisani od strane programera, te u idealnom slučaju za svaku metodu u izvornom kodu postoji referentni test koji potvrđuje njezinu funkcionalnost.

Integracijski testovi su testovi koji testiraju sučelja između dvije različite komponente, odnosno ovi testovi nam garantiraju da dvije komponente komuniciraju kako bi trebale, te da je interakcija između njih onakva kakva treba biti. Ovo je viša razina testova, te dok unit testovima testiramo da pojedini mali dio softvera radi ono za što je namijenjen, s integracijskim testovima testiramo da dva mala (ili velika) dijela softvera međusobno funkcioniraju kako bi trebala.

Sistemski testovi su testovi koji testiraju arhitekturu sustava, odnosno testiraju da neka cjelina radi na način kako je zamišljena. Takva neka cjelina može sadržavati više komponenata koje komuniciraju na razne načine, no sa sistemskim testovima možemo biti sigurni da te cjeline rade kako su zamišljene.

Testovi prihvatljivosti su testovi koji nužno ne trebaju biti provođeni od strane programera, jer za razliku od svih ostalih testova, testovi prihvatljivosti ne trebaju imati napisane testove već ručno testiranje od strane korisnika ili bilo koje druge osobe može biti dovoljno da bi se ovi testovi izvršili.

U ovome radu će se detaljnije obraditi dvije vrste testiranja, a to su integracijski i unit testovi.

5.2. Zrelost testnog procesa

Prema Borisu Beizeru, postoji određena podjela zrelosti procesa testiranja ovisno o kojem se drugačiji pristup testiranju koristi i primjenjuje u organizacijama ili pojedinim timovima. Prema njegovoj podjeli, proces testiranja ovisi o razmišljanju ljudi koji su odgovorni za testni proces, te o ciljevima zbog kojih isti i postoji. [2]

Nulti level zrelosti je način razmišljanja u kojem je testiranje jednako kao i pronalaženje pogrešaka, te se ti pojmovi poistovjećuju i ne pravi se razlika među njima. Ovaj pristup je najčešće prisutan kod timova gdje se testiranje tek uvodi ili kod pojedinaca koji se prvi puta susreću s testiranjem softvera. U ovom pristupu bi se kod testova testirali jednostavni parametri s kojima bi testirani dio koda prošao test ili ne, no ne bi se detaljno testirali svi mogući parametri.

Prvi level zrelosti je način razmišljanja gdje se s procesom testiranja pokušava dokazati da softver funkcionira. U ovome pristupu se nadograđuje nulti level, te je bolji od njega jer bi se testiralo više parametara kako bi se moglo ‘dokazati’ da neki softver radi. No problem je kod ovoga pristupa što bi se pisali testovi u kojima je poznato da softver radi, pa bi se samo s njima dobila formalna potvrda.

Drugi level zrelosti je način razmišljanja u kojemu se testnom procesu pristupa s ciljem da se dokaže da softver ne radi kako je to zamišljeno. Ovim pristupom bi se testirali parametri koji su rubni i očekivano je da možda softver ne radi najbolje s njima. Problem kod ovoga pristupa je da testove uglavnom pišu programeri, a oni će sami nesvjesno pisati testove za koje vjeruju da će proći jer da bi se sjetili parametara s kojima softver ne funkcionira, to bi se vjerojatno ispravilo u trenutku pisanja izvornog koda.

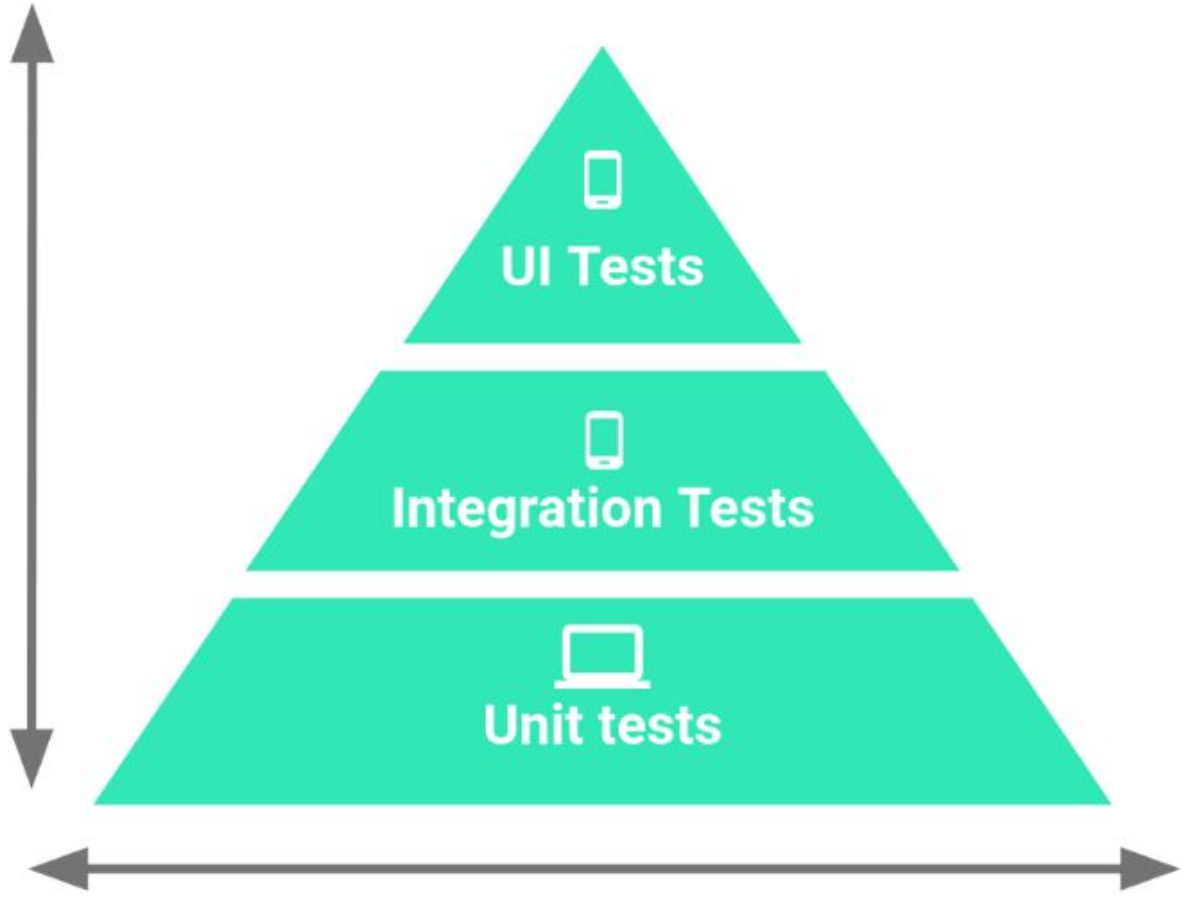
Treći level zrelosti se odnosi na način razmišljanja s kojim se pristupa procesu testiranja s ciljem da se smanji rizik kod korištenja softvera. Ovim pristupom se cijelo vrijeme ima na umu da se testiranjem može jedino potvrditi da softver pod određenim parametrima funkcionira ali se zna da to ne znači da je softver 100% bez grešaka. Cilj je

pokriti što više parametara s kojima bi se moglo pronaći pogreške kako bi cijeli tim bio sigurniji da softver u produkciji raditi korektno u što više slučajeva.

Četvrti i ujedno zadnji level zrelosti testnog procesa je proces kojemu svi teže, a to je način razmišljanja da se s testovima povećava kvaliteta softvera. Organizacije koje su na ovoj razini testiranja imaju tim testera i programera koji zajedno rade na povećanju kvalitete, gdje je uvođenje testova samo jedan mali ali bitni dio cijelog procesa.

6. Mogućnosti testiranja android aplikacija

Aplikacije za Android operacijski sustav se testiraju kroz tri različite skupine testova. Svaka skupina je nužna kako bi mogli sa sigurnošću reći da je mali rizik da nešto ne radi kako bi trebalo. Testovi se pišu prema piramidi koju možemo vidjeti na donjoj slici, a svaka kategorija se upotpunjuje s određenom skupinom testova.



Slika 4. Piramida testova za Android operacijski sustav

Uvijek se prvo testiraju najniže komponente izvornog koda, a to se radi s Unit testovima s kojima se testiraju metode u java klasama. U android testnom procesu takvi testovi se zovu mali testovi (engl. small tests). Druga kategorija su integracijski testovi s kojima se testiraju sučelja i interakcija između dvije komponente, a takvi testovi se nazivaju srednji testovi (engl. medium tests). Te na kraju nam ostaju UI testovi, odnosno veliki testovi (engl. large

tests) s kojima se testiraju UI komponente, API pozivi i slične kompliciranije radnje koje su zavisne o raznim drugim komponentama. [5]

Za provođenje unit testova nije potrebno posjedovati Android uređaj ili emulator jer se testovi izvršavaju na računalu, a zbog toga su iznimno brzi. Kod integracijskih testova je situacija nešto drugačija pa se ti testovi moraju izvršavati na Android uređaju ili na emulatoru, a ti ih čini sporim. Zadnja kategorija testova su UI testovi koji se također izvršavaju na emulatoru ili pravom Android uređaju pa zbog toga i za njih možemo reći da se izvršavaju sporo.

Iako su mali testovi brzi i efektivni, zbog različitih karakteristika testova ovisno o razini u piramidi, s njima ne možemo sa sigurnošću reći da aplikacija radi na način kako bi se trebala ponašati. Isti je problem ako naša aplikacija ima samo testove srednje ili velike razine, jer onda iako se aplikacija ponaša kako bi trebala ne možemo biti sigurni da na nižoj razini naš izvorni kod radi kako bi trebao. Iz tog razloga nije dovoljno imati napisane samo takve testove, već uvijek treba biti kombinacija malih, srednjih i velikih testova. Prema službenoj Android dokumentaciji, savjetuje se da testovi budu napisani u idućim postocima:

- 70% mali testovi
- 20% srednji testovi
- 10% veliki testovi

6.1. Testovi male razine

Najniža razina testova, odnosno testovi male razine se nazivaju unit testovi, te se izvršavaju na računalu, a služe za testiranje Java metoda. S ovim testovima možemo osigurati da metode izvornog koda rade ono za što su napisane. Dobre karakteristike ovih testova su da se izvršavaju brzo jer nije potrebno pokrenuti aplikaciju na Android uređaju ili emulatoru kako bi se izvršili. Problem kod ovih testova je da ne mogu garantirati da se aplikacija ponaša na način kako bi trebala jer iako metode rade prema specifikaciji njihova interakcija ne mora nužno značiti da je sve u redu.

Unit testovi se kod Androida trebaju pisati bez interakcije s Android razvojnim okvirom (engl Android framework), te bi trebali s njima testirati samo java kod. Naime, s obzirom na to da je u Androidu jako teško pisati Java kod bez interakcije s Android razvojnim okvirom, potrebno je u većini testova koristiti alat razvijan od Android zajednice Robolectric. On nam omogućuje da testiramo Java kod koji ima nekakvu interakciju s Android razvojnim okvirom.

Ponekada kod interakcije s Android razvojnim okvirom nam nije bitno da se kod testira u pravom okružju pa možemo koristiti razvojni okvir Mockito. Mockito sadrži android.jar koji je prazan ali smo u mogućnosti iz svoga testa pozivati metode iz Android razvojnog okvira. Iako, ako želimo da se metode iz Android razvojnog okvira izvrše u potpunosti, potrebno je koristiti Robolectric umjesto Mockita.

6.2. Testovi srednje razine

Testovi srednje razine su potrebni kako bi mogli testirati rade li pojedine komponente (koje smo već testirali s testovima male razine) u interakciji jedna s drugom na način kako je zamišljeno.

Testovi ove razine se izvršavaju sporije od testova male razine pa ih mnogi programeri izbjegavaju pisati. Ili ako ih pišu ne izvršavaju ih toliko često kao što je to slučaj s unit testovima. To je iz razloga što se ovi testovi izvršavaju na Android uređaju ili na emulatoru. Neki od primjera ovih testova su testiranje API poziva, testiranje kreiranja i spremanja nekog objekta itd.

6.3. Testovi velike razine

Najviša razina testova je ona koja testira pojedine slučajeve korištenja (engl use case), a u Android testnom procesu ih nazivamo velikim testovima. Ovi testovi su spori, ali jedini testiraju ponašanje aplikacije na razini koju će vidjeti krajnji korisnik.

U ovoj razini testova se testiraju UI komponente, pa se u puno slučajeva koristi Espresso s kojim testiramo UI komponente, stanje pojedinih komponenata te njihovu međusobnu interakciju.

7. Praktični dio rada

Za potrebe ovoga diplomskog rada, napravljena je demo aplikacija za koju su potom pisani unit, integracijski i UX testovi. Aplikacija služi za jednostavan unos zabilješki, te se naziva Noteapp, a tehnologija korištena za njezino pisanje je Java.

Funkcionalnosti aplikacije su odabrane na način da se jednostavno može prikazati postupak pisanja testova svih razina, a opet s druge strane da postoji neko upravljanje podacima i korisnicima pa da se mogu i kompleksnije stvari testirati. Fokus ovoga rada je na testovima pa je i praktični dio fokusiran uglavnom na testove, a ne kompleksnost i vizualni dio demo aplikacije.

Većina izvornog koda će biti testirana s Unit testovima, a oni služe uglavnom za testiranje čistog izvornog koda pisanog u Java programskom jeziku, te se koristi JUnit jednak koji se koristi kod testiranja Java izvornog koda za web development. S obzirom na to da se aplikacije za Android ne mogu pisati u čistoj Javi već se stalno kod isprepleće s kodom iz Android razvojnog okvira, stvari se puno kompliciraju. Iz tog razloga je praktički nemoguće uvoditi testove u postojeće aplikacije bez da se prije toga kod refakturira i napiše u boljoj arhitekturi, tj. u arhitekturi gdje je poslovna logika pisana u čistoj Javi, a preko sučelja se radi interakcija s metodama iz Android razvojnog okvira.

7.1 Opis demo aplikacije

Od funkcionalnosti aplikacija sadrži iduće:

- Registracija korisnika
- Logiranje korisnika
- Izlistavanje zabilješki
- Kreiranje nove zabilješke

Kao što je ranije navedeno, aplikacija je pisana u Javi, a arhitektura je posložena u MVP (model, view, presenter) arhitekturi. Iz tog razloga je s unit testovima potrebno testirati samo presentere jer je cijela poslovna logika u njima.

Svaka funkcionalnost se veže za posebnu komponentu u Android razvojnom okviru koja se naziva Activity, a svaki Activity uz sebe veže View i Presenter. Tako da je svaka funkcionalnost realizirana uz minimalno tri različite klase radi potrebe uvođenja testova. Da se nije radilo na taj način, bilo bi komplicirano uvoditi testove, a u mnogim slučajevima i nemoguće pa se neke ključne funkcionalnosti ne bi mogle uopće testirati.

Arhitektura u demo aplikaciji izgleda kao prema ovome primjeru. Activity implementira View sučelje, te Presenter preko njega komunicira sa metodama unutar Activitiya. Tako je omogućeno testiranje poslovne logike s čistim Java klasama i metodama, te će proces uvođenja unit testova biti jednostavan i bezbolan.

Neke funkcionalnosti imaju proširene komponente zbog potrebe za pisanjem ili čitanjem podataka iz baze podataka, ali svima je bazična struktura prema MVP paradigmi.

7.2. Uvođenje testova male razine

Kod testova ove razine se testira izvorni Java kod, te ukoliko je potrebno komunicirati s metodama iz Android razvojnog okruženja, možemo koristiti razvojni okvir Mockito [6]. On nam služi za pozivanje Android metoda ali bez njihove prave implementacije. U testovima male razine koji su pisani u ovome radu se koristio Mockito. Te je potrebno svaki test pokrenuti sa MockitoJUnitRunner klasom.

Kao što je ranije napomenuto, zbog MVP arhitekture, s testovima male razine je potrebno testirati uglavnom samo prezentere s obzirom da se u njima nalazi poslovna logika, koja se na ovoj razini testira. U aplikaciji postoje četiri prezentera, odnosno jedan prezenter za svaku funkcionalnost. Na primjeru prezentera za funkcionalnost prijave korisnika (engl. Log in) ćemo objasniti implementaciju.

Funkcionalnost prijave korisnika je realizirana preko idućih klasa:

- LoginActivity

```
public class LoginActivity extends MVPActivity<LoginPresenter> implements LoginView {

    private EditText etEmail;
    private EditText etPassword;
    private Button btLogIn;
    private ProgressBar progressBar;

    @Override
    protected void setup() {...}

    @Override
    protected int getLayout() { return R.layout.activity_login; }

    @Override
    public void onLoginSuccess() {...}

    @Override
    public void onLoginFail(String msg) {...}

    @Override
    public void onValidationSuccess() {...}

    @Override
    public void showProgress() { progressBar.setVisibility(View.VISIBLE); }

    @Override
    public void hideProgress() { progressBar.setVisibility(View.INVISIBLE); }

    @Override
    public void logIn() {...}
}
```

Slika 6. LoginActivity klasa

- LoginView

```
public interface LoginView extends View {

    void onLoginSuccess();

    void onLoginFail(String msg);

    void onValidationSuccess();

    void showProgress();

    void hideProgress();

    void logIn();
}
```

Slika 7. LoginView sučelje

- LoginPresenter

```
public class LoginPresenter extends BasePresenter<LoginView> {

    public LoginPresenter(LoginView view) { super(view); }

    public void logInPressed(EditText etEmail, EditText etPassword) {...}

    public void checkForUser(String email, String password) {...}

    public void response(User user){...}
}
```

Slika 8. LoginPresenter klasa

LoginPresenter se sastoji od nekoliko metoda, a s testovima male razine je potrebno testirati svaku dostupnu metodu, a to su redom:

- logInPressed - zove se iz LoginActivitya nakon što je korisnik pritisnuo gumb Prijavi se, a služi za provjeru jesu li polja za unos emaila i lozinke prazna ili popunjena.

```
public void logInPressed(EditText etEmail, EditText etPassword) {

    view.showProgress();
    if (ValidationUtils.isNullOrEmpty(etEmail) ||
            ValidationUtils.isNullOrEmpty(etPassword)) {
        view.hideProgress();
        view.onLoginFail( msg: "Nisu popunjena sva polja");
    } else {
        view.onValidationSuccess();
    }
}
```

Slika 9. Metoda logInPressed u LoginPresenter klasi

- checkForUser - zove se nakon što se provjera o popunjenosti polja za unos emaila i lozinke uspješno izvrši, a služi za provjeru postoji li korisnik s tim pristupnim podacima.

```
public void checkForUser(String email, String password) {
    response(Noteapp.getInstance().getDatabase().getUser(email, password));
}
```

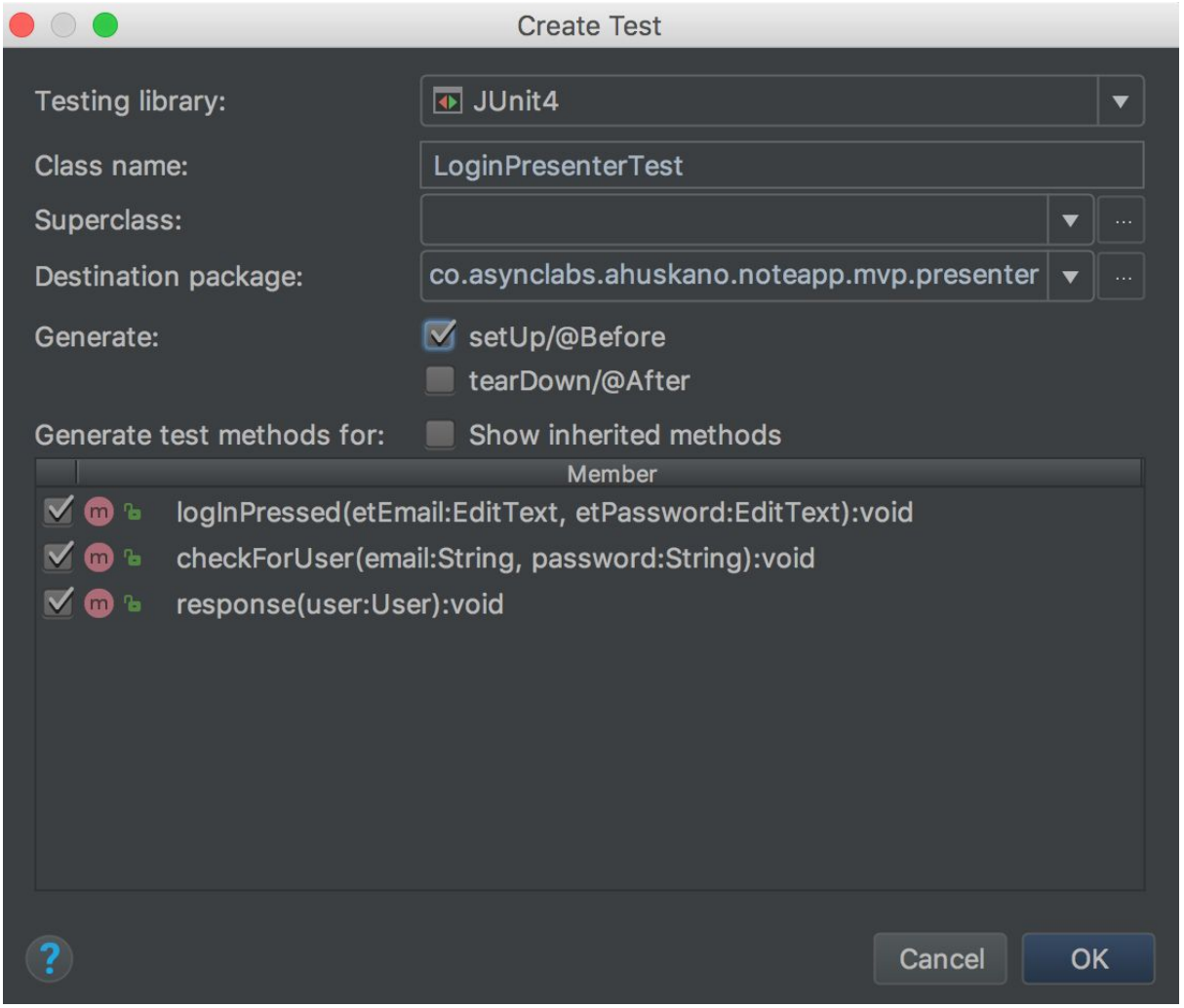
Slika 10. Metoda checkForUse u LoginPresenter klasi

- response - metoda se poziva nakon što je dohvaćen korisnik iz baze, a služi za provjeru jesu li dohvaćeni podaci ispravni ili korisnik ne postoji u bazi.

```
public void response(User user){
    if (user != null) {
        view.hideProgress();
        view.onLoginSuccess();
    } else {
        view.hideProgress();
        view.onLoginFail( msg: "Korisnik ne postoji u sustavu");
    }
}
```

Slika 11. Metoda response u LoginPresenter klasi

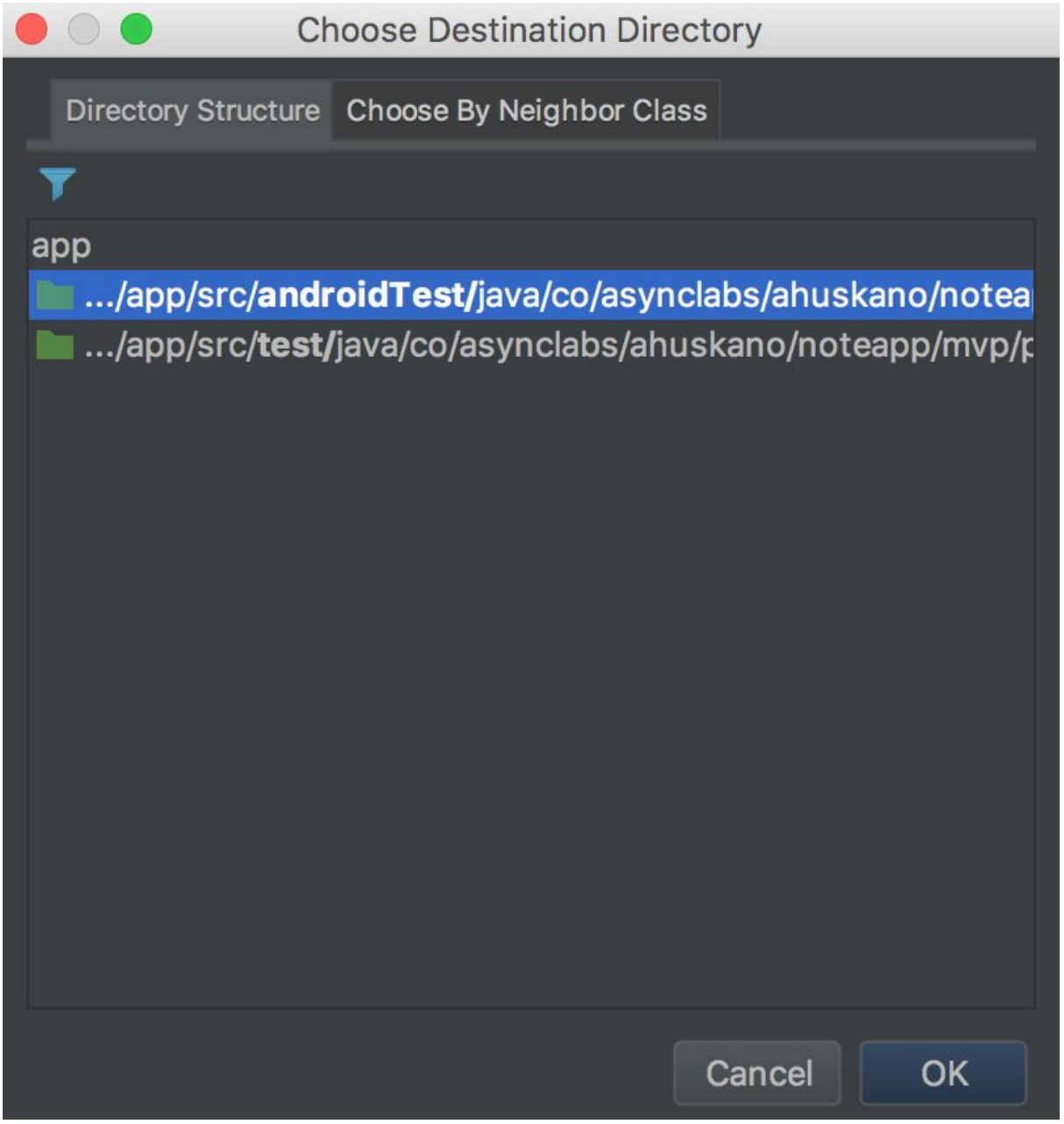
Za testove ove razine želimo da se izvršavaju na računalu, bez potrebe za Android uređajem ili emulatorom. Iz tog razloga je potrebno kreirati testne klase u direktoriju test.



Slika 12. Konfiguracijski prozor za kreiranje testne klase, korak 1

Test smo nazvali LoginPresenterTest, te smo stavili kvačice kod svake metode koje su nam ponuđene kako bi se kod generiranja klase odmah kreirale i metode gdje ćemo tada pisati testove. Uz to smo stavili kvačicu da se generira i setUp metoda, koja će se izvršiti prije pokretanja testa, a u njoj ćemo zatim inicijalizirati Mockito.

Nakon odabira postavci za generiranje, pritisnemo ok, no tada ćemo dobiti novi prozor na kojem biramo putanju gdje će se kreirati test.



Slika 13. Konfiguracijski prozor za kreiranje testa, korak 2

Imamo dvije opcije, jedna je .../app/src/androidTest/… a druga je .../app/src/test/… Odabrati ćemo drugu po redu, odnosno ovu koja ima samo test jer se testovi iz tog direktorija izvršavaju na računalo, ako odaberemo androidTest ti testovi će biti izvršeni nad Android uređajem ili emulatorom. Kod testiranja prezentera nema potrebe za izvršavanjem na pravom Android operacijskom sustavu pa ćemo odabrati samo test.

S obzirom da nije odmah Mockito uključen, potrebno ga je dodati pa iznad imena klase dodajemo @RunWith(MockitoJUnitRunner.class).

```
package co.asynclabs.ahuskano.noteapp.mvp.presenter;

import org.junit.runner.RunWith;

import org.mockito.junit.MockitoJUnitRunner;

@RunWith(MockitoJUnitRunner.class)
public class LoginPresenterTest {

}
```

Slika 14. Testna klasa LoginPresenterTest

S obzirom na to da kod funkcionalnosti prijave korisnika trebamo testirati jesu li polja za unos emaila i lozinke popunjena, prva stvar koju trebamo napraviti je mockanje potrebnih varijabli u tijelu metode setup koja će se izvršiti prije pokretanja testova. U ovom konkretnom primjeru trebamo mockati dva EditTexta te LoginView.

Mockito inicijaliziramo sa linijom koda MockitoAnnotations.initMocks(this), a kod označavanje varijabli koje želimo mockati trebamo staviti oznaku @Mock iznad njezine deklaracije. Sa tim znakom Mockito zna koje varijable neće imati pravu implementaciju već ih treba mockati.

```
@Mock
private LoginView view;

@Mock
private EditText etEmail;

@Mock
private EditText etPassword;


private LoginPresenter presenter;


@Before
public void setup() {
    MockitoAnnotations.initMocks( testClass: this);
    presenter = new LoginPresenter(view);

}
```

Slika 15. Testna klasa LoginPresenterTest sa početnom implementacijom

Prva metoda koju želimo testirati je logInPressed, s obzirom da ta metoda provjerava jesu li EditTextovi za email i lozinku popunjeni, trebamo imati dva testna slučaja. Jedan testni slučaj treba testirati izvršenje metode s nepopunjenim potrebnim podacima, dok druga metoda treba testirati ponašanje kada su oba podatka unesena.

U praktičnom primjeru je vidljivo da se u metodi logInPressed pozivaju metode iz viewa ovisno o tome jesu li polja popunjena ili ne. U slučaju da su email i lozinka uneseni, izvršila bi se metoda iz viewa onValidationSuccess(). Dok u slučaju kada neki od potrebnih podataka nije unesen, izvršavaju se hideProgress(), te onLoginFail(). U oba slučaja kao prva metoda se poziva showProgress().

Za testiranje slučaja u kojemu su email i lozinka popunjeni smo kreirali metodu logInCorrect() koja je prikazana na slici 16.

```
@Test
public void longInCorrect(){

    setupEditTexts();

    presenter.logInPressed(etEmail, etPassword);

    Mockito.verify(view, Mockito.times( wantedNumberOfInvocations: 1)).showProgress();
    Mockito.verify(view, Mockito.times( wantedNumberOfInvocations: 1)).onValidationSuccess();

}
```

Slika 16. Metoda koja testira slučaj kada je prijava uspješna

S obzirom na to da su etEmail i etPassword TextView objekti koji su mockani, potrebno je definirati što će se vratiti u slučaju da se pozove neka od metoda u njima. Za to smo iskoristili doAnswer().when() iz Mockito biblioteke.

```
private void setupEditTexts(){
    doAnswer(new Answer() {

        @Override
        public Object answer(InvocationOnMock invocation) throws Throwable {
            return new SpannableStringBuilder( text: "Email");
        }
    }).when(etEmail).getText();

    doAnswer(new Answer() {

        @Override
        public Object answer(InvocationOnMock invocation) throws Throwable {
            return new SpannableStringBuilder( text: "Password");
        }
    }).when(etPassword).getText();
}
```

Slika 17. Pomoćna metoda kod testa

Na taj način smo definirali da u slučaju da se nad mockanim objektom etEmail u slučaju da se pozove getText(), kao rezultat vrati riječ “Email”, dok u slučaju da se ista metoda pozove nad objektom etPassword vratio bi se rezultat “Password”.

Ovo je bilo nužno napraviti jer bi u suprotnom test bio neuspješan s obzirom da je ključna stvar kod provjere je li neki podatak unesen ili nije, provjera rezultata metode getText() nad EditText objektom.

Sa Mockito.verify(mockani objekt, broj izvršavanja).nazivMetode() smo testirali jesu li se izvršile obje metode iz prezentera, a to su showProgress() te onValidationSuccess().

Nakon što smo testirali poslovnu logiku kod uspješne provjere parametara za login, potrebno je testirati i slučaj u kojem email ili password nisu dobro upisani. Za taj test smo napravili dvije različite metode kako bi pokazali dva primjera na kojemu se može testirati ista funkcionalnost.

```
@Test
public void logInWrong1() {
    InOrder inOrder = inOrder(view);

    presenter.logInPressed( etEmail: null,  etPassword: null);
    inOrder.verify(view).showProgress();
    inOrder.verify(view).hideProgress();
    inOrder.verify(view).onLoginFail(any(String.class));
}
```

Slika 18. Metoda koja testira neuspješnu prijavu

U metodi logInWrong1 smo koristili InOrder koji nam omogućuje testiranje redoslijeda metoda koje se izvršavaju [7]. S obzirom da želimo biti sigurni da se progress bar pokaže prije nego li se bilo kakva provjera izvrši, InOrder iz Mockita je izvstan odabir za ovakav test.

```
InOrder inOrder = inOrder(firstMock, secondMock);

inOrder.verify(firstMock).add("was called first");
inOrder.verify(secondMock).add("was called second");
```

Slika 18. Struktura InOrder poziva iz Mockito biblioteke

InOrder je sučelje Mockita koji se koristi ako želimo testirati nad mockanim objektom redoslijed metoda kojim se one izvršavaju. Osim toga on nam omogućuje i da testiramo koliko puta se neka metoda izvršila, te je li se neka metoda uopće izvršila. Prema slici 18

možemo vidjeti da se može kombinirati više mockanih objekata nad kojima i dalje možemo pratiti redoslijed kojim su se metode izvršavale.

```
InOrder inOrder = inOrder(firstMock, secondMock);

inOrder.verify(firstMock, times(2)).someMethod("was called first two times");
inOrder.verify(secondMock, atLeastOnce()).someMethod("was called second at least once");
```

Slika 19. Naprednije struktura InOrder poziva iz Mockito biblioteke

Kao što je ranije napomenuto InOrder sučelje nam omogućuje i praćenje koliko puta se koja metoda pozvala, te prema gornjoj slici vidimo da postoje parametri sučelje s kojima se može to konfigurirati. S metodom times(broj interakcija) možemo konkretno testirati bilo koliki broj poziva, dok u slučaju da metoda treba biti pozvana najmanje jedanput za to bi koristili atLeastOnce. Na kraju je uvijek dobro pozvati i inOrder.onMoreInteraction() jer s njim testiramo da neće biti pozvane još neke metode iz mockanih objekata, a time osiguravamo da nam neće promaknuti neki poziv funkcije koju nismo namjeravali pozvati.

U našem testu smo pozvali metodu prezentera logInPressed sa null objektima kao parametrima, te smo zatim pratili hoće li se metode iz viewa pozvati idućim redoslijedom: showProgress, zatim hideProgress te tek na kraju onLoginFail. Ovaj test će biti uspješan jedino ako su metode pozvane baš tim redoslijedom, te će u bilo kojem drugom slučaju (uključujući slučaj u kojem neka od metoda nije pozvana uopće) biti neuspješan.

```
@Test
public void logInWrong2() {

    presenter.logInPressed( etEmail: null,  etPassword: null);
    Mockito.verify(view, Mockito.times( wantedNumberOfInvocations: 1)).showProgress();
    Mockito.verify(view, Mockito.times( wantedNumberOfInvocations: 1)).hideProgress();
    Mockito.verify(view, Mockito.times( wantedNumberOfInvocations: 1)).onLoginFail(any(String.class));
    Mockito.verify(view, Mockito.never()).onLoginSuccess();
}
```

Slika 20. Drugi način testiranja neuspješne prijave

Sa metodom logInWrong2 smo koristili drugi pristup, te u njoj nismo testirali redoslijed izvršenja metoda već smo testirali broj puta koliko je neka metoda pozvana. Tako smo ponovno pozvali metodu prezentera logInPressed s null objektima, te smo pratili hoće li se metode showProgress, hideProgress te onLoginFail izvršiti točno po jedanput. Za dodatnu

provjeru smo koristili Mockito.never s kojim možemo biti sigurni da se niti u jednom slučaju kod poziva metode logInPressed s null objektima neće izvršiti metoda onLoginSucess.

U primjeru dva smo sigurniji da se neće pozvati ono što nije potrebno (onLoginSucess) ali nismo sigurni kojim redoslijedom će se pozvati metode. Na primjer, u prvom primjeru se mogla izvršiti nakon metode onLoginFail metoda onLoginSucess, a naš bi test bi bio uspješno izvršen i ta pogreška se ne bi uspjela njime detektirati, dok se u drugom primjeru to ne bi moglo dogoditi.

Iduće je potrebno testirati što se dogodi u poslovnoj logici nakon što se vrati rezultat provjere korisnika, odnosno kada se pozove metoda response() iz prezentera. Metoda će u slučaju da je uspješno pronađen korisnik koji se pokušava ulogirati pozvati metode onHideProgress i onLoginSuccess iz viewa, dok će u suprotnom pozvati onHideProgress, te onLoginFail sa porukom da korisnik nije pronađen.

Tu metodu smo odlučili testirati sa Mockitom preko InOrder funkcionalnosti. S obzirom da je nama važno da se progress ugasi prije nego li se korisniku ispiše poruka o uspješnom/neuspješnom logiranju, odabrali smo ovaj način.

```
@Test
public void response(){
    InOrder inOrder = inOrder(view);
    presenter.response( user: null);
    inOrder.verify(view).hideProgress();
    inOrder.verify(view).onLoginFail(any(String.class));

    presenter.response(new User());
    inOrder.verify(view).hideProgress();
    inOrder.verify(view).onLoginSuccess();
}
```

Slika 21. Testiranje metode response iz LogInPresenter klase

Na slici 21 se može vidjeti implementacija testne metode, u prvom slučaju se testira ponašanje prezentera u slučaju da je kao parametar primio null, odnosno da korisnik nije pronađen, dok je u drugom slučaju poslan objekt tipa User koji metodi govori da je korisnik pronađen. Ovaj test će biti neuspješan u slučaju da se metode izvrše krivim redoslijedom, ili

ako se neka od njih uopće ne izvrši. S obzirom da se provjera oba slučaja izvršava sekvencijalno, metode se moraju izvršiti baš ovim redoslijedom kojim se vrši provjera pa smo test za oba slučaja ostavili u jednoj metodi umjesto razdvojili u dvije različite.

7.3. Uvođenje testova srednje razine

Nakon uvođenja testova male razine, odnosno unit testova, dolazi do procesa uvođenja testova srednje veličine. Za razliku od malih testova, ovi testovi testiraju međuzavisnosti komponenata, te kod Androida testiraju interakciju java koda sa Android razvojnim okvirom. Kao što smo ranije napomenuli da kod unit testova se testira samo java code, te treba izbjegavali bilo kakvu interakciju sa Android razvojnim okvirom. S obzirom na to da je iznimno teško razdvojiti java kod od Android koda, testovi srednje veličine se izvršavaju na uređaju ili an emulatoru, a služe baš za testiranje interakcije s klasama iz Android razvojnog okvira.

Kod implementacije testova srednje veličine, u našem praktičnom primjeru, koristili smo Espresso biblioteku. Kao što je ranije napomenuto, ona nam omogućuje da testiramo UI komponente, njihova stanja, te interakciju između njih.

Kod kreiranja ovog tipa testa, potrebno je odabrati u izborniku androidTest putanju, umjesto test putanje koju smo birali kod kreiranja unit testova. Ovisno na kojoj putanji je testna klasa kreirana ovisi hoće li se izvršavati na računalu ili na emulatoru/uređaju.

U ovome radu će biti prikazana implementacija testa srednje veličine nad klasom LoginActivity. Ova klasa je odabrana iz razloga što ima najviše funkcionalnosti s kojima se može prikazati svrha i implementacija testa srednje razine. Za prvi korak kreirana je klasa LoginActivityTest.

```
@MediumTest
@RunWith(AndroidJUnit4.class)
public class LoginActivityTest {

    LoginActivity activity;

    @Rule
    public IntentsTestRule<LoginActivity> mActivityRule
            = new IntentsTestRule<>(LoginActivity.class);

    @Before
    public void setUp() throws Exception {
        activity = mActivityRule.getActivity();
    }

}
```

Slika 22. LogInActivity testna klasa

Prema slici 22 je vidljivo da za razliku od Unit testova, koje smo pokretali sa JunitRunnerom, ovdje testove pokrećemo sa AndroidJunit4.class, te je iznad stavito @MediumTest dok toga nije bilo kod prethodnih testova.

Koristili smo IntentsTestRule jer preko njega Espresso testira i radi interakciju sa Intentovima, a s obzirom da mi u ovom testu testiramo LoginActivity, nad njim smo i inicijalizirali Rule.

Na sučelju od LoginActivitya postoje tri elementa: dva elementa EditText gdje je jedan za email, te drugi za lozinku te Button s kojim se započinje akcija prijave korisnika. To je prva stvar koju smo odlučili testirati, a metoda u kojoj smo testirali se naziva setup iz razloga što je to inicijalna provjera nalaze li se svi potrebni elementi na ekranu.

```
@Test
public void setup() {
    onView(withId(R.id.etEmail)).check(matches(isDisplayed()));
    onView(withId(R.id.etPassword)).check(matches(isDisplayed()));
    onView(withId(R.id.btnLogin)).check(matches(isDisplayed()));
}
```

Slika 23. Konfiguracija parametara iz LogInActivity testne klase

Na slici 23 možemo vidjeti da su tri linije implementacije gotovo iste, jedina razlika u njima je id od elementa. Metoda onView iz Espressa nam omogućuje da pozovemo neki view iz activityje koje testiramo, a sa metodom withId smo odabrali način mapiranja preko jedinstvenog identifikatora, osim toga mogli smo koristiti neki drugi način no da ne kompliciramo odabrali smo jednostavno id. Sa metodom check() radimo određenu provjeru, a vidljivo je u implementaciji da smo koristili matches te zatim kao parametar proslijedili isDisplayed(). Na taj način smo osigurali da nam test prođe jedino u slučaju da EditTextovi sa emailom i lozinkom i Button za akciju prijave budu prikazani na zaslonu, odnosno u layoutu koji se koristi kod LoginActivity klase.

Iduća metoda koju je potrebno testirati je onLoginFail(), ta metoda se poziva u slučaju da prijava korisnika nije uspješna te je potrebno korisniku sa Toast porukom ispisati na ekran kako je njegova prijava neuspješna.

```
@Test
public void onLoginFail() {

    activity.runOnUiThread(new Runnable() {
        public void run() {
            activity.onLoginFail(activity.getString(R.string.login_failed_msg));
        }
    });
    onView(withText(activity.getString(R.string.login_failed_msg))).
            inRoot(withDecorView(not(is(activity.getWindow().getDecorView())))).
            check(matches(isDisplayed()));
}
```

Slika 24. Testiranje neuspješne prijave

Pri testiranju ove metode smo ponovno koristili Espresso no s drugačijim metodama. Na slici 24 se može vidjeti da smo pozvali metodu onLoginFail ali u UI dretvi, jer da smo je zvali sekvencijalno u dretvi u kojoj se izvršava test dobili bi pogrešku jer se testovi ne izvršavaju u UI dretvi. Ponovno smo koristili onView s obzirom da testiramo je li nešto prikazano na ekranu ili ne, ali ovoga puta nismo tražili element na ekranu preko jedinstvenog identifikatora s obzirom da je ovo Toast i dinamički mu se svaki puta pridruži id već smo koristili withText. WithText je metoda u Espressu koja nam omogućuje da na prikazanom dijelu ekrana pronađemo traženi element samo na temelju teksta koji u njemu treba biti prikazan, a mi smo iskoristili poruku koja se prikazuje u slučaju da prijava korisnika bude neuspješna.

Nakon što korisnik pritisne gumb za akciju prijave, treba se prikazati progress bar, a nakon izvršene provjere isti se treba prekinuti. S obzirom da se to događa kod interakcije sa najbitnijom funkcionalnosti ovoga activitya, odlučili smo i to testirati.

```
@Test
public void showProgress() {

    activity.runOnUiThread(new Runnable() {
        public void run() {
            activity.showProgress();
        }
    });

    Drawable notAnimatedDrawable = ContextCompat.getDrawable(activity, R.drawable.ic_launcher_background);
    ((ProgressBar) activity.findViewById(R.id.progressBar)).setIndeterminateDrawable(notAnimatedDrawable);

    onView(withId(R.id.progressBar)).check(matches(isDisplayed()));

}
```

Slika 25. Testiranje prikaza progress bara

Na slici 25 je prikaz implementacije showProgress gdje smo ponovno koristili Espresso. Naime, Espresso ima problem sa ProgressBar elementom, te da bi test ispravno radio potrebno je na element postaviti Drawable (koji nigdje neće biti prikazan ali je bitno da je na elementu). Tek nakon što smo to napravili Espresso će nam ispravno provjeriti je li ProgressBar prikazan ili nije, u suprotnom ne bi mogao to napraviti.

Osim showProgress, postoji i metoda hideProgress koju smo testirali an identičan način, a implementacija je vidljiva na slici 26.

```
@Test
public void hideProgress() {

    activity.runOnUiThread(new Runnable() {
        public void run() {
            activity.hideProgress();
        }
    });

    Drawable notAnimatedDrawable = ContextCompat.getDrawable(activity, R.drawable.ic_launcher_background);
    ((ProgressBar) activity.findViewById(R.id.progressBar)).setIndeterminateDrawable(notAnimatedDrawable);

    onView(withId(R.id.progressBar)).check(matches(not(isDisplayed())));
}
```

Slika 26. Testiranje skrivanja progress bara

Zadnja metoda koju smo testirali sa testom srednje veličine je logIn, odnosno metoda koja se zove nakon što su podaci za email i lozinku provjereni i valjani. Prema njezinoj implementaciji, ta metoda će pozvati i pokrenuti novi Intent odnosno, novi Activity. U ovom primjeru smo testirali je li se nakon poziva te metode pokrenuo Activity koji bi trebao.

```
@Test
public void logIn() {
    activity.logIn();
    intended(toPackage(LoginActivity.class.getPackage().getName()));
}
```

Slika 27. Testiranje uspješne prijave

LogIn metodu smo testirali sa intended() također iz Espresso biblioteke koja nam omogućuje da provjerimo je li pozvan Activity iz određenog paketa. Naravno ovaj test odgovara našem primjeru no u slučaju da su svi Activity spremljeni pod istim paketom, ovaj test bi uvijek bio uspješan pa ne bi znali je li pokrenut baš onaj Activitiyi koji treba biti ili neki drugi.

7.4. Uvođenje testova velike razine

Nakon uvođenja testova male i srednje razine, potrebno je uvesti testove velike razine. Takvi testovi se ujedno i najsporije izvršavaju, ali testiraju najkompleksnije dijelove aplikacije. Odnosno s ovim testovima testiramo cijele slučajeve korištenja, te međusobnu interakciju između nekoliko aktivnosti. Iako smo s ranijim razinama testova testirali pojedine metode i aktivnosti, jedino s velikim testovima možemo biti sigurni da će se sve dobro izvršiti.

Kako izgleda neki test velike razine prikazati ćemo na testiranju interakcije između dvije funkcionalnosti. Test će provjeriti može li se korisnik koji se upravo registrirao u aplikaciju prijaviti s podacima novo kreiranog korisnika.

```
@LargeTest
public class SignUpTest {

    @Rule
    public IntentsTestRule<SignUpActivity> mActivityRule
            = new IntentsTestRule<>(SignUpActivity.class);


}
```

Slika 28. Klasa za testiranje registracije korisnika

Na slici 28 je prikazana klasa SignUpTest u kojoj ćemo napraviti našu implementaciju testa velike razine. Ponovno smo koristili IntentsTestRule iz Espresso biblioteke kako bi imali pristup do elemenata na tom Activityu.

```
@Test
public void registration() {

    onView(withId(R.id.etName)).perform(typeText( stringToBeTyped: "Alen"));
    onView(withId(R.id.etEmail)).perform(typeText( stringToBeTyped: "alen@email.com"));
    onView(withId(R.id.etPassword)).perform(typeText( stringToBeTyped: "password"));

    Espresso.closeSoftKeyboard();

    onView(withId(R.id.etPasswordRepeat)).perform(typeText( stringToBeTyped: "password"));

    Espresso.closeSoftKeyboard();

    onView(withId(R.id.btnRegistration)).perform(click());

    onView(withId(R.id.etEmail)).perform(typeText( stringToBeTyped: "alen@email.com"));
    Espresso.closeSoftKeyboard();

    onView(withId(R.id.etPassword)).perform(typeText( stringToBeTyped: "password"));
    Espresso.closeSoftKeyboard();

    onView(withId(R.id.btnLogin)).perform(click());

    onView(withId(R.id.rvNotes)).check(matches(isDisplayed()));

}
```

Slika 29. Metoda koja testira registraciju i prijavu korisnika

U metodi registration() smo napravili test koji ispuni formu za registraciju korisnika, te zatim pokuša napraviti prijavu tog novokreiranog korisnika. Prvo smo u svaki EditText u formi upisali potrebne podatke, pa smo napravili pritisak na gumb registracije. Nakon toga, u demo aplikaciju bi se upalio novi Activity koji služi za prijavu korisnika, odnosno LogInActvity. U njemu smo zatim unijeli iste te podatke koje smo unijeli pri registraciji te smo pritiskom na gumb login izvršili prijavu. Iako smo to sve napravili, još ne možemo biti sigurni je li se korisnik uistinu i prijavio, te smo zbog toga dodali provjeri je li RecyclerView rvNotes prikazan na ekranu, ako je to znači da je otvoren DashboardActivity, a on se prikazuje jedino korisnicima koji su uspješno prijavljeni.

S testovima ove razine je bitno biti na oprezu jer činjenica da slučaj korištenja gdje se novo registrirani korisnik može prijaviti ne znači da je isti pravilno spremljen u bazu podataka ili da se pravilno odvila prijava korisnika. Za takve stvari nam služe testovi nižih razina, te je s njima potrebno testirati sve manje funkcionalnosti pa ako su one zadovoljene možemo bez straha i iznenađenja reći da ovaj slučaj korištenja funkcionira upravo onako kako bi trebao.

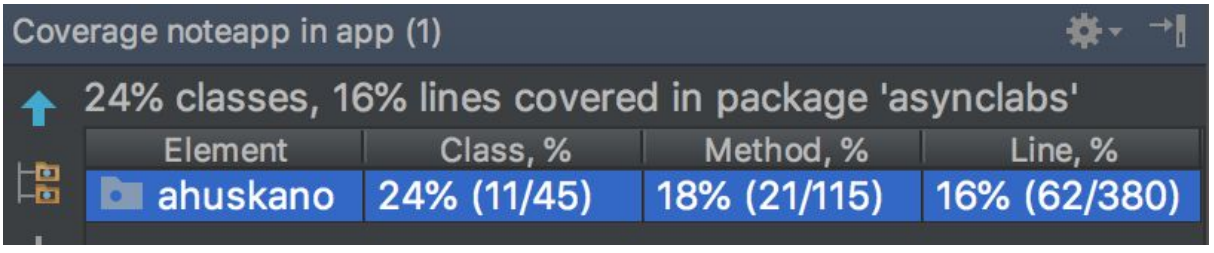
7.5. Pokrivenost testovima demo aplikacije

Sa svakim testiranjem za cilj imamo smanjenje rizika od nepravilnog izvršavanja funkcionalnosti aplikacije, no ono što je najbitnije je način na koji možemo pratiti kolika je pokrivenost izvornog koda testovima. Android Studio nam omogućuje da izvršimo testove s praćenjem postotka pokrivenosti. Nažalost ta pokrivenost podrazumijeva jedino Unit testove, odnosno testove male razine, dok za testove srednje i velike razine ne dobijemo nikakav sumirani prikaz pokrivenosti.

Pokrivenost testovima se može pratiti u tri različite razine: pokrivenost klasa, pokrivenost linija koda, te pokrivenost metoda. S obzirom na ove kategorije, očito je da je praktički nemoguće postići 100% pokrivenost testovima, jer bi to značilo da baš svaku metodu i liniju koda trebamo imati testiranu, a kod velike većine koda nema smisla pisati testove pa bi to samo bilo vremensko preopterećenje i loše korištenje resursa.

Demo aplikacija rađena za potrebe ovoga rada ima iduće pokrivenosti:

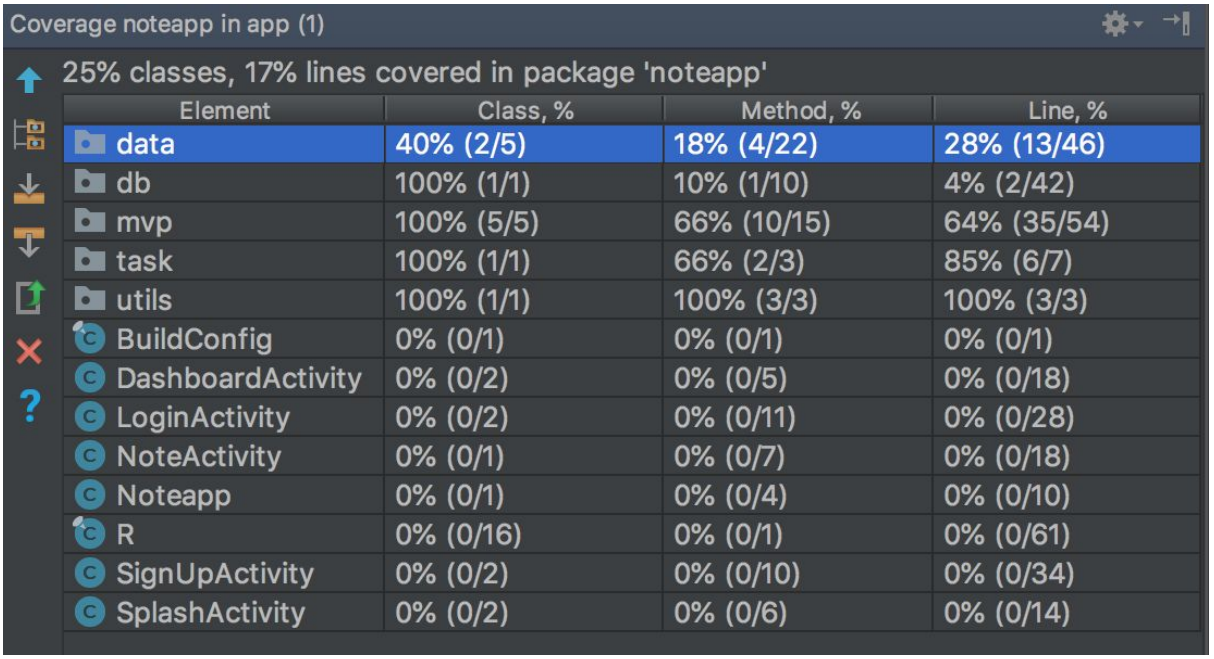
- Pokrivenost klasa: 24%
- Pokrivenost linija: 18%
- Pokrivenost metoda: 16%



Slika 30. Sumirani prikaz pokrivenosti unit testovima

Na slici 30 je prikazana pokrivenost cijelog direktorija gdje se nalazi izvorni kod za aplikaciju, no ako pogledamo detaljniji prikaz na slici ispod primijetit ćemo da su mnoge klase kao što su Activity klase ostale s pokrivenosti 0%. To je iz razloga što su te klase zbog interakcije sa

Androdi razvojnim okvirom testirane sa testovima srednje i velike razine, a kao što je ranije navedeno, Android Studio prikazuje pokrivenost samo Unit testova.



Slika 31. Detaljni prikaz pokrivenosti unit testovima

Postoje određene biblioteke koje mogu prikazati pokrivenost testova s uključenim testovima srednje i velike razine, no one nisu bile predmet ovoga rada jer je sve korišteno za njegovu izradu službeno od strane Googlea ili prihvaćeno od Googlea kao standard, iako su pisani od treće strane.

8. Zaključak

U modernom procesu razvoja softvera uvođenje proces testiranja je prijeko potrebno i više ne može biti ignorirano. Sve kompanije koje drže do sebe pišu testove, a startupi koji razvijaju softverski proizvod s testovima održavaju kvalitetu cijeloga proizvoda. U android aplikacijama je testove puno teže uvesti nego u web aplikacijama no ne znači da je nemoguće. Istina da je teško pokriti Android aplikaciju sa Unit testovima, no u kombinaciji unit, integracijskih i UI testova odnosno u kombinaciji testova male, srednje i velike razine moguće je testirati gotovo svaku funkcionalnost. Iz tog razloga ne postoji opravdanje ili razlog za ne pisanje testova za Android aplikacije jer su se biblioteke kao što su Espresso i Mockito razvile do te razine da ih je vrlo jednostavno koristiti.

U androidu je bitno pisati testove sve razine jer kao što je vidljivo iz praktičnog primjera u ovome radu, testovi na jednoj razini testiraju samo jednu razinu interakcije, te je jedini način za kompletno i stabilno uvođenje testova uvođenje svih razina. Ako se netko odluči na samo jednu razinu, testovi će uvijek biti manjkavi zbog nemogućnosti testiranja ostalih razina interakcije.

9. Literatura

1. Beck, K. 2000. Extreme Programming Explained. Addison- Wesley
2. Boris Beizer. Software Testing Techniques. Van Nostrand Reinhold, New York, 2nd edition, 1990. ISBN 0-442-20672-0.
3. State Counter, Mobile Operating System Market Share Worldwide, preuzeto 24.8.2018. http://gs.statcounter.com/os-market-share/mobile/worldwide
4. Developer Android, Distribution dashboard, preuzeto 24.8.2018. https://developer.android.com/about/dashboards
5. Developer Android, Testing fundamentals, preuzeto 24.8.2018. https://developer.android.com/training/testing/fundamentals
6. Mockito JUnitRunner documentation, preuzeto 3.9.2018. https://static.javadoc.io/org.mockito/mockito-core/2.6.8/org/mockito/junit/MockitoJUnitRunner.html
7. Mockito InOrder documentation, preuzeto 5.9.2018. https://static.javadoc.io/org.mockito/mockito-core/2.6.9/org/mockito/InOrder.html
8. Izvorni kod demo projekta, https://github.com/ahuskano/NoteApp